CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Kyoto University Research Information Repository
KURENAI
Kyoto University Research Information Repository
KYOTO UNIVERSITY

Title	Some Analysis of PASCAL Programs (Mathematical Methods in Software Science and Engineering)
Author(s)	SHIMASAKI, MASAAKI; FUKAYA, SHIGERU
Citation	数理解析研究所講究録 (1979), 363: 41-58
Issue Date	1979-09
URL	http://hdl.handle.net/2433/104570
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

Kyoto University

数理解析研究所講究録
363巻 1979年 41-58

Some Analysis of PASCAL Programs

Masaaki Shimasaki
Department of Information Science
Kyoto University

Shigeru Fukaya
Hankyu Corporation

1. Introduction

Recently necessity and importance of measurement and analysis of actual programs have been recognized in programming language study[1-5]. The aim of the approach is as follows:

(1) Utilizing statistical results, some characteristics of a programming language and/or a programming methodology can be made clear. Results can be hopefully feed back to designers of programming languages, translators and high level language oriented machines to improve programming environment.

(2) A result, especially dynamic behavior of a program is useful to a programmer for better program development. Thus a program analysis system plays a role of a software tool for program development and improvement.

We have implemented a PASCAL program analysis system[6]. The second version of the analysis system has been designed and is now under implementation. There may be two kinds of analysis, i.e. static and dynamic. A method and some results of static analysis of PASCAL programs are given in section 2. In section 3, a method of dynamic analysis is described. In section 4, some experiments on code improvement of Sequential PASCAL programs are discussed.

2. Static analysis of PASCAL programs and profile of PASCAL compilers

Static analysis is carried out by recording occurrences of various statements of source language as they appear within user's program. The result of analysis reflects how a language is used in an actual program text. For gathering static profiles of programs, both manual and automatic approaches are used. In automatic approach, there may be two ways;

(1) to develop a special syntax analyzer for this purpose

and

(2) to modify an existing compiler.

Since a PASCAL compiler is usually written in PASCAL itself, it is quite easy to insert statements for gathering static information.

In the second version of our PASCAL program analysis system, we have decided to carry out 'control flow analysis' at the level of source statements. We partition statements into 'maximal' groups such that no transfer occurs into a group except to the first statement in the group. It is obvious that once the first statement in a group is executed, all the statements in that group are executed sequentially. We call such a group of statements a 'block'. In order to carry out control flow analysis in recursive descent fashion, we adopt some 'worst case assumption' and the 'maximality' of a block in our definition is based on this assumption; in our control flow analysis, we assume that if there is a statement label excluding case label, then there are at least two statements transferring to that label. Generally speaking, a labeled statement should be the first statement of a block. If there is only one statement transferring to that label, then the labeled statement can be possibly included in a larger block. Even in such a case, we assume that a block is partitioned by the labeled statement. If a procedure is referred to only one procedure call statement, then the procedure call statement and the first statement of the procedure can possibly belong to the same block. We again assume, however, that even in such a case, a block is partitioned by a procedure call statement.

the same block. We again assume, however, that even in such a case, a block is partitioned by a procedure call statement.

It should be noted that some type of statement is always the first statement of a block and some type of statement is always the last statement of a block. Under our assumption, following properties can be utilized to partition statements into blocks.

(1) A goto statement is the last statement of a block.
(2) A procedure call statement is the last statement of a block.
(3) A labeled statement is the first statement of a block.
(4) A if statement is the last statement of a block. Thus a statement following to then or an else clause becomes the first statement of another block. The situation is similar with a case statement.
(5) A for statement is the first statement of a block.
A repeat statement is the first statement of a block.
A while statement is the first statement of a block.

Blocks are numbered and the number of a block is determined by the compilation order in recursive descent compilation. It should be noted that if we check 'in degree' of each node in the control flow graph after compiling all the statements, we can construct blocks with exact 'maximal' property. If there is any block with 'indegree of 1', then the block cane be appended to its preceeding block.

The control flow analysis may be utilized for 'dead code elimination', though it is not expected that such a case occurs frequently in PASCAL programs. The result of the control flow analysis is utilized by dynamic analysis described in section 3.

PASCAL is a language designed for use in stepwise refinement programming. We are interested in how PASCAL is actually used, especially in the field of compiler writing. We now describe the result of a case study of static analysis on two existing PASCAL compilers written in PASCAL[6]. PASCAL compilers under investigation are:

(1) a standard PASCAL compiler for FACOM 230-38[7], a descendant from a compiler developed at ETH.
(2) a Sequential PASCAL compiler extracted from the SOLO system

developed by Per Brinch Hansen[8] and adapted to HITAC 8350 computer, an IBM 360 type computer. Hereafter we designate them by S_t and S_q, respectively. The two compilers have the following characteristics:

(i) They are fairly complex system programs. They are designed carefully in the top-down stepwise refinement approach and may be considered well-structured programs.

(ii) They are system programs written in a high level programming language. It is expected they will reveal characteristics of system programs, especially of compilers, written in a high level language.

(iii) S_t is a one-pass, in-core compiler. S_q is, on the other hand, a multi-pass (7-pass) compiler utilizing a disc as auxiliary storage, They may be typical instances of compilers.

Specifications of the two languages are slightly different and statistical results are described separately. Statement distribution in the PASCAL compilers is given in Table 1. Approximately one-third to one half of all the statements are procedure call statements. According to reports on FORTRAN[1,3] and on PL/I[4], the percentage of CALL statements was between 2 percent and 8 percent in both FORTRAN samples and in PL/I samples, which was much lower than that in our PASCAL samples. Table 2 shows statistics about how many times procedures are referred to statically in the source program text. About one half of all the procedures are referred to only once in the source program text. This is the case both with S_t and with S_q. This result shows a role of procedures in stepwise refinement programming. It should be noted that a procedure which is referred to at only one place of source program text cannot be a recursive procedures. There is no difficulty with inline substitution of such a procedure. The information about with of such procedures is invoked dynamically for many times is useful from the view point of program optimization by inline substitution of procedures. Thus we recognize the necessity of a software tool for obtaining dynamic behavior of procedures.

Percantage of *if* statements is approximately the same as those in FORTRAN [1,3] and in PL/I[4]. *Goto* statements are rarely used in PASCAL. Table 3 shows how many *if* statements have *else* clause. About 50 percent of *if* statements have *else* clause. The result is interesting in view of Elshoff's statement in his paper on analysis of PL/I programs[4]: " Only 17 percent of the IF statements in the programs analyzed had an ELSE clause. Sometimes it is very reasonable to not have an ELSE caluse but not that often. Hand analysis indicates that somewhere between 50 percent and 80 percent of the IF statements should have ELSE clauses if the language is properly used...."

While , *repeat* , *for* and *case* statements are less frequently used than the *if* statement.

Table 1 Distribution of statement types in two PASCAL compilers

	S_t		S_q (7-pass)	
	NO	%	NO	%
PROGRAM	1		7	
PROCEDURE	146	3.0	465	8.4
FUNCTION	3		12	
PROCEDURE FORWARD	1		48	0.86
FUNCTION FORWARD	0		0	
LABEL	14	0.29	/	
CONST	1		19	
TYPE	3		13	
VAR	79	1.6	192	3.5
Assignment	1763	36.0	1333	24.0
Proc. Call (user)	1508	30.8	2302	41.4
Proc. Call (standard)	75	1.6	24	0.43
Proc. Call (I/o, system)	34	0.73	192	3.45
IF	926	18.9	567	10.2
WITH	160	3.7	193	3.47
WHILE	59	1.2	27	0.49
REPEAT	46	0.93	49	0.88
FOR	32	0.65	61	1.1
CASE	30	0.61	56	1.0
GO TO	17	0.35	/	
Total	4898		5560	

Table 2 Number of static references of procedures

number of static reference	number of procedures in percent	
	S_t	S_q
1	53.4%	48.1%
2	9.6	13.5
3	4.8	8.2
4-10	15.1	19.5
11-	17.1	10.7

Table 3 If statement and else clause

	S_t	S_q
number of if statements	926	567
number of else	493	336
percent ratio	53.2%	59.3%

3. Dynamic analysis of PASCAL programs

Dynamic analysis is investigation of dynamic behavior of a program. A software system to feedback a programmer information about the execution profile of his program is very useful for the purpose of improving program efficiency. For FORTRAN, results of dynamic analysis are reported in [1,2]. Ingalls[2] says: " in a typical program only 3% of the statements make up 50% of program's execution time. All the programs which I have seen since fall into the same pattern, and I would expect it to be about the same for most languages also." In such a case monitoring the execution profile is quite useful for detecting a program bottleneck and optimizing it or improving an algorithm. Thus as one fundamental tool for program development, a language processor system should provide a user with a software facility for monitoring execution of his program. Stucki[5] went further and introduced the concept of automated tools for improving software quality.

We have implemented PASDAP[6], a system for gathering the dynamic behavior of a PASCAL programs. PASDAP is a preprocessor. It takes a PASCAL program as input, and outputs an equivalent PASCAL program which also maintains frequency counts of invokation of procedures and writes them on to a file. When the converted PASCAL program is compiled and run, we will get the number of invokations of each procedure together with usual result of execution.

In PASDAP, measurement is carried out for procedures and functions rather than for statements. The strategy made PASDAP simple to a great extent and runtime overhead of PASDAP is kept small. Evaluation of execution time or running cost of each procedure/function may be carried out by a hardware clock or by some software method. The feasibility of timing by a hardware clock depends greatly on the environment. We implemented PASDAP system on HITAC 8350 computer system, and its one-second timer is of little use for the purpose. The Sequential PASCAL compiler translates a PASCAL program into virtual stack machine code and the object machine code is executed by an interpreter. If the weight

for each virtual stack machine instruction is determined, it is easy to construct a virtual software clock by summing up weights of machine instructions executed by the interpreter. PASDAP uses the virtual clock for time measurement. Table 4 shows the result of dynamic analysis of Pass 1 of S_q, while compiling Pass 1 of S_q itself. This pass is rather I/O bound and drastic improvement of efficiency cannot be expected. PASDAP was applied to analysis of PASDAP itself and was very useful for its improvement.

Besides static distribution of statement types, runtime distribution of statement types is also very interesting. We have designed PASDAP-II which carries out measurement on distribution of statement types executed and statement execution counts, besides counting numbers of procedure calls and procedure timing. As mentioned earlier, runtime overhead of measuring statement counts will be very large, if we adopt a method like one used in PASDAP. Instead of counting number of execution of each statement, PASDAP-II counts number of executions of each 'block'. In the static analysis, we keep records of a statement type for each statement and the number of the block which each statement belongs to. Statement execution counts and distribution of statements types executed can be computed from 'block' execution counts, utilizing the result of static analysis. Thus the static analysis will reduce overhead of dynamic analysis to a great extent. PASDAP-II is not a preprocessor but a part of a compiler and dynamic analysis is a compiler option.

Table 4 An example of dynamic analysis by PASDAP

S	PROCEDURE NAME	D	WEIGHT
1	MAIN	1	$8.9*10^{4}$
1	INIT_PASS	1	0.8
1	NEXT_PASS	1	2.1
8	WRITE_IFL	9885	$4.6*10^{3}$
1	LARGEST_REAL	1	1.1
3	PUT_ARG	262	$4.8*10^{2}$
18	PUT0NC	1899	$3.9*10^{3}$
10	PUT0	1294	$2.4*10^{3}$
8	PUT1	3215	$1.2*10^{4}$
1	PUT_STRING	50	$1.3*10^{3}$
48	STD_ID	48	$4.5*10^{2}$
1	STD_NAMES	1	$5.2*10^{2}$
5	END_LINE	1006	$1.1*10^{4}$
4	GET_CHAR	138	$1.3*10^{3}$
1	INIT_OPTIONS	1	$4.2*10^{2}$
1	INITIALIZE	1	$1.5*10^{3}$
1	NUMBER	232	$2.1*10^{3}$
1	SAME_ID	2719	$2.0*10^{3}$
1	INSERT_ID	285	$3.6*10^{2}$
1	SEARCH_ID	2412	$8.6*10^{3}$
3	STRING_CHAR	1034	$7.0*10^{2}$
1	STRING	288	$5.4*10^{3}$
1	IDENTIFIER	2412	$4.2*10^{4}$
1	SCAN	1	$8.7*10^{4}$

S : Number of static reference
D : Number of invokations

4. Some experiments on code improvement of Sequential PASCAL programs [9]

4.1 Code improvement by inline substitution of procedures

As investigation in preceding sections shows, it is often the case that a program is composed by many small procedures in stepwise refinement programming. It is therefore expected that 'inline' substitution' of a procedure body for a procedure call statement improves runtime efficiency. If a procedure is referred to at many places in a source program, inline substitution of such a procedure for all the call statements causes increase in memory space required. Therefore it is necessary to select 'appropriate' procedures for inline substitution. In this context, PASDAP system is useful for selection of procedures but we try to set a criterion of automatic selection by a compiler. The criterion we set is to satisfy all the following conditions:

(a) A procedure which is referred to at only one place in a source program text.

(b) A procedure which is not declared as 'forward'.

(c) A procedure which does not have a 'univ' type parameter.

Reasons why we set the condition (a) are as follows: Since such a procedure cannot be a recursive procedure, inline substitution of such a procedure is relatively easy. If a procedure is referred to only once, inline substitution of such a procedure does not increase memory space required. Finally there are many procedures satisfying the condition in the S_q compiler.

The condition(b) was set because a preliminary investigation showed that most of procedures declared as forward were referred to more than two times and many of them were recursive procedures. Table 5 shows the result of the preliminary investigation of procedures in the S_q compiler.

The S_q compiler is a 7-pass compiler* and treats 8 kinds of languages including source, intermediate and object languages. The intermediate code produced by the compiler is a seqence of integers. Each integer represents either an operator in the intermediate language, or an argument of an operator. We decided to carry out inline substitution of procedures for the output language of Pass 3; By inline substitution of a procedure,

---

* Pass 1 : Lexical Analysis, Pass 2 : Syntax Analysis
Pass 3,4,5 : Semantic Analysis, Pass 6,7 : Code Generation

interprocedural program optimization is reduced to intraprocedural program optimization and it is therefore preferable to carry out inline substitution at an earlier stage of compilation. At the end of Pass 3, syntax analysis and name analysis have been done but allocation of memory space to each variable has not been done. It is therefore easy to introduce new variables for inline substitution at the end of Pass 3.

An algorithm of inline substitution of a procedure is as follows:

(1) If a 'called' procedure to be expanded has any locally declared type or variable, such a declaration is to be transferred to the global area.

(2) For 'call-by-value' formal parameters in a 'called' procedure to be expanded, corresponding new variable declarations are inserted in the global area.

(3) If a 'called' procedure to be expanded has a 'call-by-value' formal parameter, then an assignment statement of the value of the actual parameter(expression) to the corresponding newly declared variable is inserted before expanding the procedure body of the 'called' procdeure.

(4) A 'call-by-reference' formal parameter is replaced by the corresponding actual parameter variable.

(5) A procedure call statement is replaced by the corresponding body of the 'called' procedure.

Fig. 1 gives an example of inline substitution of procedures. Although actual expansion is carried out using an intermediate language, the example is shown in the source program for clarity. It should be noted that in the example, procedure expansion of the procedure b is interrupted by the procedure expansion of procedure a. The procedure expansion is carried out in a nested form.

In our implementation, procedure inline expansion is carried out using 3 phases. In the first phase, each procedure is determined to be substituted or not. At the same time, necessary information for procedure expansion

Table 5 Summary of Procedures in the S_q Compiler

	PASS 1	PASS 2	PASS 3	PASS 4	PASS 5	PASS 6	PASS 7
(1)	32	68	104	81	75	49	44
(2)	15	6	53	41	33	19	21
(3)	17	33	54	41	33	19	21
(4)	18	62	47	32	29	28	32
(5)	5	4	6	3	4	10	13

(1) Procedures in total
(2) Procedures which can be expanded
(3) Procedures referred to only once
(4) Procedures with parameters
(5) Procedures with parameters which can be expanded

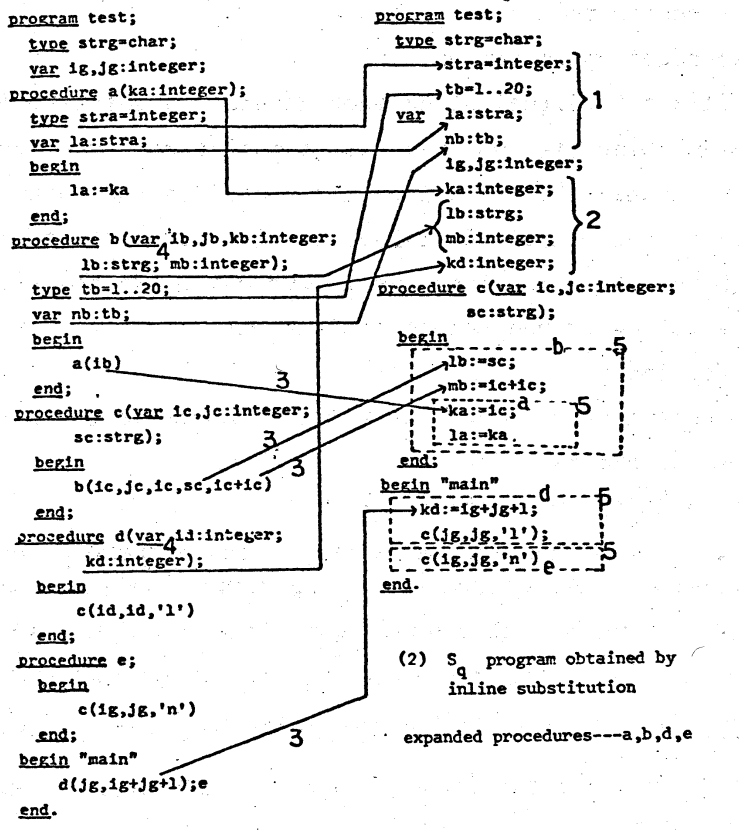

Fig. 1 An example of inline substitution of procedures

is also collected. In the second phase, the systematic renaming of name indecies is carried out. In the third phase, a procedure call statement is replaced by the corresponding procedure body.

4.2 Some peephole optimizations

Peephole optimization is said to be effective with a relatively small amount of effort and some kind of peephole optimization is carried out in almost all compilers. Among peephole optimization techniques often used are:

i) elimination of unnecessary load or store instruction to or from registers
ii) constant folding
iii) dead code elimination
iv) operator strength reduction
v) machine dependent optimization

The S_q system utilizes a virtual stack machine and its software interpreter. The virtual stack machine does not have usual registers for evaluation of expressions. Thus usual register allocation technique for optimization cannot be applied to our case. It is not expected that the technique of operator strength reduction is effective in our case. We therefore, decided to carry out the following peephole optimizations:

(1) constant folding
If all the operands are constant values, the value of the expression is computed at compile time.

(2) machine dependent optimization
The virtual machine has special instructions increment and decrement which increments and decrements by 1, respectively , the value of a 'for-loop' control variable whose address is on the stack top. Extending definition of these instructions, we use these instructions not only for 'for-loop' control variable but for usual variables.

Peephole optimizations may cause changes in code length, and peephole optimizations should be carried out in advance of address calculation such

as jump address calculation. Pass 5 of the S_q compiler is the body analysis and Pass 6 and Pass 7 are code selection and code assembly, respectively. The peephole optimization ia applied to output codes of Pass 5 in our implementation.

We define 21 delimiting operators out of 49 operators in the output language of Pass 5. They delimit block of codes to which the peephole optimization is applied. Among them are 'deflabel', 'jump' , 'assgin', etc. Operators such as 'add' , 'pushconst' are not delimiting operators.

Peephole optimization by the use of 'increment' and 'decrement' operator is carried out in the following way. The technique can be applied to the case when a delimiting operator is 'assign' operator and its operand is 1. The operand '1' implies that the operator is applied to 'word' type. Code patterns to which the optimization technique can be applied are shown in Fig.2. If code pattern under consideration is one of patterns shown in Fig. 2, then the block of code is replaced by corresponding optimized codes.

4.3 Experimental results of code improvement

Inline substitution of procedures and peephole optimization were applied to 4 test programs and improvement of execution time was evaluated. Execution time or cost was evaluated by a virtual software clock technique used in PASDAP, summing up weights of virtual machine code executed by the interpreter. Four test programs are as follows:

(1) A program designed to print the first 999 prime numbers greater than 2 (54 lines). The program was quoted from Wirth[10] and a routine for conversion of an integer to a character string was added. The additional routine was also quoted from the S_q compiler.

(2) A program to generate a sequence of 20 characters, chosen from an alpfhabet of three elements, such that no two immediately adjacent subsequences are equal. (69 lines) This program was also quoted from Wirth[10].

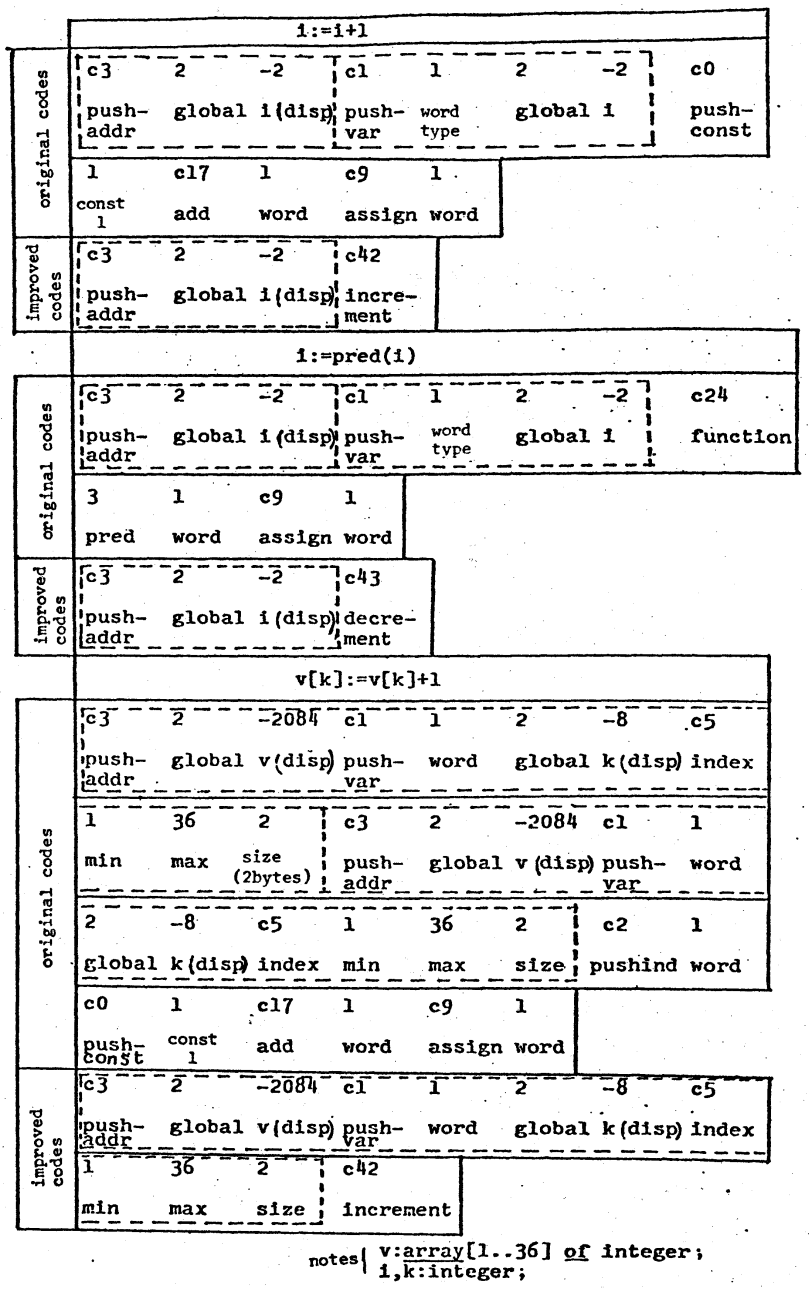

Fig. 2 Example code patterns of peephole optimization

(3) Pass 1 of the S_q compiler (the lexical analysis pass). (1006 lines)

(4) Pass 3 of the S_q compiler (the name analysis pass). (1862 lines)

Programs (1) and (2) are examples of short programs, and programs (3) and (4) are examples of relatively large programs. Table 6 shows execution time or cost of these programs. In Table 6, four results are given for each program. They are:

(a) the result of the original program,

(b) the result of a program obtained by 'inline substitution' of procedures,

(c) the result of a program obtained by peephole optimizations,

(d) the result of a program obtained by inline substitution of procedures and by peephole optimizations.

In Table 6, inputs to programs (3) and (4) are programs (3) and (4), respectively. Table 7 gives results of programs (3) and (4) by changing input data. Judging from these results, results given in Table 5 might be considered as typical ones. In these examples, the time for execution of 'callsys' operator , i.e. the time for input and output operations is excluded.

Scheifler[11] carried out inline substitution of procedures for programs written in CLU and evaluated improvement in execution time and change in memory space. According to his results, improvements in execution time were 28%, 5%, 17% 13% and 7%. He says that " execution time saved directly by inline substitution is small, even for fairly inefficient procedure call mechanisms; however, the enlarged context made available to other techniques may lead to much more optimization than would otherwise be possible." The effect of inline substitution of procedures in our examples is smaller than that in Sceifler's examples. Reasons might be as follows: In our examples, the S_q system uses a virtual stack machine system and the overhead of procedure call and return on the virtual machine is relatively small. Therefore, if implementation of procedure call and return mechanism is

Table 7 Result of code improvement (2)

Input Data Program			Test Program: Lexical Analysis Program original	Lexical Analysis Program improved	Name Analysis Program original	Name Analysis Program improved
Character Sequence Generator		A: codes executed	125,269	120,851	61,242	57,060
		B: callsys executed	10,428	equal to left	32	equal to left
		A - B	114,851	110,423	61,210	57,028
		C: cost total	5,138,206.31	4,974,628.79	2,321,275.71	2,186,636.83
		D: cost callsys	746,123.43	equal to left	2,289.60	equal to left
		C - D	4,392,082.88	4,228,505.36	2,318,986.11	2,184,347.23
		cost improved	0.00	163,577.52	0.00	134,638.88
		percentage	0.00	3.72	0.00	5.81
Character Sequence Generator (Compiled)		A: codes executed	238,245	225,968.	176,430	166,284
		B: callsys executed	14,883	equal to left	5,265	equal to left
		A - B	223,362	211,085	171,165	161,019
		C: cost total	9,399,876.59	8,943,907.49	6,680,709.90	6,365,877.79
		D: cost callsys	1,064,878.70	equal to left	376,710.77	equal to left
		C - D	8,334,997.89	7,879,028.79	6,303,999.13	5,989,167.02
		cost improved	0.00	455,969.10	0.00	314,832.11
		percentage	0.00	5.47	0.00	4.99
Lexical Analysis Program		A: codes executed	1,616,309	1,544,242	809,667	739,351
		B: callsys executed	151,987	equal to left	266	equal to left
		A - B	1,464,322	1,392,255	809,401	739,085
		C: cost total	67,070,483.02	64,489,829.46	31,036,402.30	28,706,549.62
		D: cost callsys	10,874,670.31	equal to left	19,032.30	equal to left
		C - D	56,195,812.71	53,615,159.15	31,017,370.00	28,687,517.32
		cost improved	0.00	2,580,653.56	0.00	2,329,852.68
		percentage	0.00	4.59	0.00	7.51
Name Analysis Program		A: codes executed	3,668,241	3,538,253	1,526,364	1,401,516
		B: callsys executed	281,293	equal to left	526	equal to left
		A - B	3,386,948	3,256,960	1,525,838	1,400,990
		C: cost total	151,541,807.69	146,892,252.78	58,127,597.01	53,989,922.69
		D: cost callsys	20,126,515.01	equal to left	37,635.30	equal to left
		C - D	131,415,292.68	126,765,737.77	58,089,961.71	53,952,287.39
		cost improved	0.00	4,649,554.91	0.00	4,137,674.32
		percentage	0.00	3.54	0.00	7.12

Table 6 Result of code improvement (1)

		original	inline substitution	peephole optimizations	inline sub. + peephol opts.
Prime Number Generator	statements	54			
	A: codes executed	1,089,446	1,088,447	1,002,572	1,001,573
	B: callsys executed	5,994	equal to left	equal to left	equal to left
	A - B	1,083,452	1,082,453	996,578	995,579
	C: cost total	39,392,930.07	39,283,639.48	36,769,335.36	36,660,044.76
	D: cost callsys	428,870.72	equal to left	equal to left	equal to left
	C - D	38,964,059.35	38,854,768.76	36,340,464.64	36,231,174.04
	cost improved	0.00	109,290.59	2,623,594.71	2,732,885.31
	percentage	0.00	0.28	6.73	7.01
Character Sequence Generator	statements	69			
	A: codes executed	1,275,156	1,259,067	1,137,783	1,121,694
	B: callsys executed	8,756	equal to left	equal to left	equal to left
	A - B	1,266,400	1,250,311	1,129,027	1,112,938
	C: cost total	47,453,733.77	46,536,392.68	43,302,773.46	42,385,432.37
	D: cost callsys	626,491.83	equal to left	equal to left	equal to left
	C - D	46,827,241.94	45,909,900.85	42,676,281.63	41,758,940.54
	cost improved	0.00	917,341.09	4,150,960.31	5,068,301.40
	percentage	0.00	1.96	8.86	10.82
Lexical Analysis Program	statements	1,006			
	A: codes executed	1,616,309	1,599,405	1,561,146	1,544,242
	B: callsys executed	151,987	equal to left	equal to left	equal to left
	A - B	1,464,322	1,447,418	1,409,159	1,392,255
	C: cost total	67,070,483.02	66,106,644.29	65,453,668.19	64,489,829.46
	D: cost callsys	10,874,670.31	equal to left	equal to left	equal to left
	C - D	56,195,812.71	55,231,973.98	54,578,997.88	53,615,159.15
	cost improved	0.00	963,838.73	1,616,814.83	2,580,653.56
	percentage	0.00	1.72	2.88	4.59
Name Analysis Program	statements	1,862			
	A: codes executed	1,526,364	1,506,822	1,421,058	1,401,516
	B: callsys executed	526	equal to left	equal to left	equal to left
	A - B	1,525,838	1,506,296	1,420,532	1,400,990
	C: cost total	58,127,597.01	57,112,551.68	55,004,968.03	53,989,922.69
	D: cost callsys	37,635.30	equal to left	equal to left	equal to left
	C - D	58,089,961.71	57,074,916.38	54,967,332.73	53,952,287.39
	cost improved	0.00	1,015,045.33	3,122,628.98	4,137,674.32
	percentage	0.00	1.75	5.38	7.12

appropriate, then the overhead of procedure call and return is not so large, even if there are many procedures in stepwise refinement programming.

The improvement in execution time attained by constant folding was less than 0.2% therefore negligible in our examples.

It is interesting that the extended use of 'increment' and 'decrement' operators results improvement in execution time by 2.8%- 8.8%. Those operators were originally designed for changing 'for-loop' control variables. The result shows that code patterns, 'increment by 1' and 'decrement by 1' frequently occur not only for 'for-loop' variables but for usual variables. It can be expected that investigation on 'code patterns' and introduction of new high level operators may make further improvement in execution time.

5. Concluding remarks

Static and dynamic analysis methods of a PASCAL program are discussed. It is preferable to utilize static analysis to reduce overhead of dynamic analysis. A static profile of PASCAL compilers is shown. The result is different from those of FORTRAN and PL/I reported before. It indicates the impact of programming discipline on program structure. Experimental results of code improvement by inline substitution and by peephole optimizations were also discussed.

Acknowledgements

Authors express their sincere gratitude to Emeritus Professor T. Kiyono of Kyoto University and Professor K. Ikeda at the University of Tsukuba for their valuable discussions at the early stage of the project. Acknowledgements are due to Professor Per Brinch Hansen and Mr. M. Takeichi for kindness of supplying us with their PASCAL system tapes.

References

[1] D. Knuth: An Empirical Study of FORTRAN Programs, Software-Practice & Experience, Vol. 1, pp. 105-133(1971).

[2] D. Ingalls: The Execution Time Profile as a Programming Tool, in Design and Optimization of Compilers, pp.107-128, Prentice-Hall, New Jersey, 1972.

[3] S. K. Robinson and I. S. Torsun: An Empirical Analysis of FORTRAN Programs, Computer Journal, Vol. 19, pp.56-62(1976).

[4] J. L. Elshoff: An Analysis of Some Comercial PL/I Programs, IEEE Trans. on Software Engineering, Vol. SE-2, pp.113-120 (1976).

[5] L. G. Stucki: New Directions in Automated Tools for Improving Software Quality, in Trends in Programming Methodology, Vol. II Program Validation, pp.81-111, Prentice-Hall, New Jersey, 1977.

[6] M. Shimasaki, S. Fukaya, K. Ikeda, T. Kiyono: A PASCAL Program Analysis System and Profile of PASCAL Compilers, Proceedings of the Twelfth Hawaii International Conference on System Sciences, Vol. I, Selected Papers in Software Engineering and Mini-Micro Systems, pp.85-90 (1979).

[7] M. Takeichi: PASCAL Compiler for the FACOM 230 OS2/VS, Dept. of Mathematical Engineering, University of Tokyo, 1975.

[8] P. Brinch Hansen: The SOLO Operating System, Software-Practice & Experience, Vol.6, pp.141-205(1976).

[9] S. Fukaya: On Code Improvement of Sequential PASCAL Programs, Master's Thesis, Dept. of Information Science, Kyoto University, 1979.

[10] N. Wirth: Systematic Programming: An Introduction, Prentice-Hall, 1973.

[11] R. M. Scheifler: An Analysis of Inline Substitution for a Structured Programming Language, Comm. ACM Vol. 20, pp.647-654 (1977).